Appendix A

Travel.cpp

Attorney Docket 3296.1

Inventor: Earl A. Hubbell


```
#include <cstring.h>
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define TAB '\t'
#define ARRAYHASH 256


char base_from_num(long num)
{
        switch (num%4)
        {
                case 0: return('A');
                case 1: return('C');
                case 2: return('G');
                case 3: return('T');
                default: return ('A');
        }
}

long transform(string &Test)
{
        Test.to_upper();
        for (long i=0; i<Test.length(); i++)
   {
        if (Test[i]=='A')
        Test[i]='T';
                else
              if (Test[i]=='C')
                Test[i] = 'G';
         else
                      if (Test[i]=='G')
                        Test[i]='C';
            else
                              if (Test[i]=='T')
                                                Test[i]='A';
        }
   return(TRUE);
}


class ProbeSynth{
```

```
public:
        string Probe;
        string Original;
        string Name;
        string Rename;
        long Location;
        long Unit;
        long Atom;
        // done with fancy
        char *Synth;
        long SynthLength;
        // shift by 4,8,12
        long SynthModifier;
   long MutVal; // mutation position
        ProbeSynth();
   -ProbeSynth();
   Destroy();
        Allocate(long);
   Synthesize();
   SynthesizeFluff(long);
   SynthesizeFastFluff(long);
   SynthesizeMutant(long);
   SynthesizeMisMatch(long);
   CompSynth();
   char GetSynth(long);
        SetSynth(long, char);
   ZeroCost();
   Distance(ProbeSynth *);
        Output(ofstream &);
        List(ofstream &);
};

ProbeSynth::SynthesizeMutant(long MutateValue)
{
        long synthflag, done, fwdflag;
        long probecount, cyclecount, qloop;
   long start, finish;
        long resynth;

   start = 0;
        finish = 0;
   Allocate(4*Probe.length()+4);
        probecount = 0;
        this->MutVal = MutateValue;
        cyclecount = start;
        done = FALSE;
        synthflag = FALSE;
        fwdflag = TRUE;

        while (!done)
        {
             if (probecount==MutateValue)
             {
                  if (probecount<Probe.length()-1)
                  {
                       resynth = cyclecount;
                       cyclecount=resynth+5; /*leave room for mutant*/
                       /*synthesize mutant*/
                       synthflag = FALSE;
                       for (qloop=resynth; qloop<cyclecount-1; qloop++)
                       {
                            if(base_from_num(qloop)==Probe[probecount])
                            {
                  // let us know that this is a mutant (and hence different)
             // >if< we were doing ACGT, filling in these with all 4
             // would let us match probes well, since geographic position
             // would matter.  But we're only doing single mismatches,
             // for GE, so that doesn't work.
                                 SetSynth(qloop, '*');
                                 synthflag = TRUE;
                            }
```

```
                        else
                SetSynth(qloop, '*');
                        }
                        if (!synthflag)
                                done = TRUE;
                        probecount++;
        cyclecount--;
                        /*cancel out cyclecount++ later*/
                }
                else
                {
                        if (base_from_num(cyclecount)== Probe[probecount])
                        {
                                SetSynth(cyclecount, base_from_num(cyclecount));
                                synthflag = TRUE;
                                probecount++;
                        }
                }
                if (probecount>Probe.length()-1)
                        done = TRUE;

                if (finish>0 && synthflag)
                {
                        fwdflag = FALSE;
                        probecount = Probe.length()-1;
                        cyclecount = finish;
                }
                else
                {
           cyclecount++;
                }
           }
           else
                if (base_from_num(cyclecount)==Probe[probecount])
                {
                        if (fwdflag)
                        {
                                SetSynth(cyclecount, base_from_num(cyclecount));
                                probecount++;
                                cyclecount++;
                                if (probecount>=Probe.length())
                                        done = TRUE;
                        }
                        else
                        {
                                SetSynth(cyclecount, base_from_num(cyclecount));
                                probecount--;
        cyclecount--;
                                if (probecount<=MutateValue)
                                        done = TRUE;
                        }
                }
                else
                {
                        if (fwdflag)
        cyclecount++;
                        else
        cyclecount--;
                }
        }
        this->SynthLength = cyclecount; // shorten 'search' space
        return(TRUE);
}

ProbeSynth::SynthesizeMisMatch(long MutateValue)
{
        long synthflag, done, fwdflag;
        long probecount, cyclecount, qloop;
        long start, finish;
        long resynth;
```

```
    start = 0;
    finish = 0;
        Allocate(4*Probe.length()+4);
        probecount = 0;
        cyclecount = start;
    this->MutVal = MutateValue;
        done = FALSE;
        synthflag = FALSE;
        fwdflag = TRUE;

        while (!done)
        {
                if (probecount==MutateValue)
                {
                        if (probecount<Probe.length()-1)
                        {
                                resynth = cyclecount;
                                cyclecount=resynth+5; /*leave room for mutant*/
                                /*synthesize mutant*/
                                synthflag = FALSE;
                                for (qloop=resynth; qloop<cyclecount-1; qloop++)
            {
                                        if(base_from_num(qloop)==Probe[probecount])
                                        {
                                                // let us know that this is a mutant (and hence
different)
                                                SetSynth(qloop, '#');
                                                synthflag = TRUE;
                                        }
            }
                                if (!synthflag)
                                        done = TRUE;
                                probecount++;
            cyclecount--;
                                /*cancel out cyclecount++ later*/
                        }
                        else
                        {
                                if (base_from_num(cyclecount)== Probe[probecount])
                                {
                                        SetSynth(cyclecount, base_from_num(cyclecount));
                                        synthflag = TRUE;
                                        probecount++;
                                }
                        }
                        if (probecount>Probe.length()-1)
                                done = TRUE;

                        if (finish>0 && synthflag)
                        {
                                fwdflag = FALSE;
                                probecount = Probe.length()-1;
                                cyclecount = finish;
                        }
                        else
                        {
                                cyclecount++;
                        }
                }
                else
                        if (base_from_num(cyclecount)==Probe[probecount])
                        {
                                if (fwdflag)
                                {
                                        SetSynth(cyclecount, base_from_num(cyclecount));
                                        probecount++;
            cyclecount++;
                                        if (probecount>=Probe.length())
                                                done = TRUE;
                                }
                                else
```

```
                        {
                                SetSynth(cyclecount, base_from_num(cyclecount));
                                probecount--;
                cyclecount--;
                                if (probecount<=MutateValue)
                                        done = TRUE;
                        }
                }
                else
                {
                        if (fwdflag)
                cyclecount++;
                        else
                                cyclecount--;
                }
        }
        this->SynthLength = cyclecount; // shorten 'search' space
        return(TRUE);
}

char
ProbeSynth::GetSynth(long Which)
{
        if (Which < SynthLength && Which>-1)
                return(Synth[Which]);
        else
                return('.');
}

ProbeSynth::SetSynth(long Which, char What)
{
        if (Which>-1 && Which < SynthLength)
        Synth[Which] = What;
   return(TRUE);
}

ProbeSynth::ZeroCost()
{
        return(Probe.length());
}

ProbeSynth::Distance(ProbeSynth *Destination)
{
        static char testchar;
        static long count;
        static long minlen;
        static long i;

        count = 0;
        minlen = min(Destination->SynthLength, this->SynthLength);

        for (i=0; i<minlen; i++)
        {
                testchar = Destination->Synth[i];  // can guarantee i>0
                if (testchar!='.')
                        if (this->Synth[i]!=testchar)
                        {
                                if (testchar == '#')
                                        count+=2;  // heavily penalize mutants
                                else
                                        count++;
                        }
        }
        for (; i<Destination->SynthLength; i++)
        {
                testchar = Destination->Synth[i];  // can guarantee i>0
                if (testchar!='.')
                {
                        if (testchar == '#')
                                count+=2;  // heavily penalize mutants
                        else
```

```
                        count++;
                }
        }
        return(count);
}

ProbeSynth::Synthesize()
{
        long Size = CompSynth();
        Allocate(Size);
        long length, probeloop, synthloop;
        char testchar;

        length = Probe.length();
        probeloop = 0;
        synthloop = 0;
        while (probeloop<length)
        {
                testchar = base_from_num(synthloop);
                if (testchar==Probe[probeloop])
                {
                        Synth[synthloop] = Probe[probeloop];
                        probeloop++;
                }
                else
                        Synth[synthloop] = '.';
                synthloop++;
        }
        return(TRUE);
}

ProbeSynth::SynthesizeFluff(long MutateValue)
{
        long length, probeloop, synthloop, testloop;
        char testchar;

        length = Probe.length();
        long Size = 4*length+4;
        Allocate(Size);
   this->MutVal = MutateValue;
        probeloop = 0;
        synthloop = 0;
   for (probeloop=0; probeloop<length; probeloop++)
        {
                synthloop= probeloop*4;
      testchar = Probe[probeloop];
        for (testloop = 0; testloop<4; testloop++, synthloop++)
                {
                    if (base_from_num(synthloop)==testchar)
          {
                if (probeloop!=MutateValue)
                                Synth[synthloop] = testchar;
                        else
                Synth[synthloop] = '#';
          }
                        else
                Synth[synthloop] = '.';
      }
        }
        return(TRUE);
}

ProbeSynth::SynthesizeFastFluff(long MutateValue)
{
        long length, probeloop, synthloop, testloop;
        char testchar;

        length = Probe.length();
        long Size = length+1;
        Allocate(Size);
   this->MutVal = MutateValue;
```

```
        probeloop = 0;
        synthloop = 0;
    for (probeloop=0, synthloop = 0; probeloop<length; probeloop++, synthloop++)
        {
                Synth[synthloop] = Probe[probeloop];
       if (probeloop==MutateValue)
       {
        synthloop++; // account for mutant being twice as bad
          Synth[synthloop]=Probe[probeloop];
                }
        }
        return(TRUE);
}

ProbeSynth::CompSynth()
{
        long length, probeloop, synthloop;
        char testchar;

        length = Probe.length();
        probeloop = 0;
        synthloop = 0;
        while (probeloop<length)
        {
                testchar = base_from_num(synthloop);
                if (testchar==Probe[probeloop])
                {
                        probeloop++;
                }
                synthloop++;
        }

        return(synthloop);
}


ProbeSynth::ProbeSynth()
{
        Probe = "";
   Name = "None";
        Location = 0;
   Synth = NULL;
        SynthLength = 0;
        SynthModifier = 0;
}

ProbeSynth::Allocate(long Size)
{
        Destroy();
   Synth = new char [Size];
   for (long i=0; i<Size; i++)
                Synth[i] = '.';
   SynthLength = Size;
}

ProbeSynth::Destroy()
{
        if (Synth!=NULL)
   {
        delete[] Synth;
      Synth = NULL;
      SynthLength = 0;
   }
}

ProbeSynth::~ProbeSynth()
{
        Destroy();
}

ProbeSynth::Output(ofstream &OutStream)
```

```
{
        OutStream << this->Location << TAB;
        OutStream << this->Original << TAB;
        OutStream << this->Name << TAB;
        OutStream << this->Rename << TAB;
        OutStream << this->Unit << TAB;
        OutStream << this->Atom << endl;
}

ProbeSynth::List(ofstream &OutStream)
{
        OutStream << this->Original << endl;
}

class ProbeNode{
public:
        ProbeSynth *DataPointer;
   ProbeNode *Previous;
   ProbeNode *Next;
        ProbeNode **PClosest;
        long plength;
        long marker;
        long NextCost;
        PointToData(ProbeSynth *);
        ProbeNode();
   ~ProbeNode();
   Destroy();
   DestroyPClosest();
        DestroyData();
   AllocatePClosest(long);
        LinkPrevious(ProbeNode *);
   LinkNext(ProbeNode *);
   InsertNext(ProbeNode *);
   Initialize();
        ZeroCost();
   Copy(ProbeNode *);
   Distance(ProbeNode *);
};

ProbeNode::Copy(ProbeNode *Original)
{
        if (Original!=NULL)
                DataPointer = Original->DataPointer;
        else
                return(FALSE);
   // note - do not copy Pclosest, plength - don't apply to copies
   // do not copy previous & next, because they won't correspond.
}


ProbeNode::PointToData(ProbeSynth *Data)
{
        DataPointer = Data;
}

ProbeNode::ZeroCost()
{
        return(DataPointer->ZeroCost());
}

ProbeNode::Distance(ProbeNode *Destination)
{
        return(DataPointer->Distance(Destination->DataPointer));
}

ProbeNode::ProbeNode()
{
        DataPointer = NULL;
   Previous = NULL;
   Next = NULL;
        PClosest = NULL;
```

```
    marker = 0;
        plength =0;
    NextCost = 0;
}

ProbeNode::Destroy()
{
        DataPointer = NULL;
    Previous = NULL;
    Next = NULL;
}

ProbeNode::DestroyPClosest()
{
        if (PClosest !=NULL)
        {
        delete[] PClosest;
       PClosest = NULL;
       plength = 0;
        }
}

ProbeNode::DestroyData()
{
        // dangerous - must be last called
    if (DataPointer!=NULL)
    {
                delete[] DataPointer;
       DataPointer = NULL;
        }
}

ProbeNode::AllocatePClosest(long Size)
{
        DestroyPClosest();
        PClosest = new ProbeNode * [Size];
  for (long i=0; i<Size; i++)
        PClosest[i] = NULL;
    return(TRUE);
}

ProbeNode::~ProbeNode()
{
        Destroy();
}

ProbeNode::LinkPrevious(ProbeNode *Link)
{
        Previous = Link;
}

ProbeNode::LinkNext(ProbeNode *Link)
{
        Next = Link;
}

ProbeNode::InsertNext(ProbeNode *NewNode)
{
        ProbeNode *Link;

    Link = Next;
        Next = NewNode;
    NewNode->Previous = Link->Previous;
    Next->Previous = NewNode;
    NewNode->Next = Link;
}

ProbeNode::Initialize()
{
        Previous = this;
    Next = this;
```

```
}

class TourClass{
public:
        ProbeNode *DataList;
   long Cost;
        TourClass();
        InitializeTour(ProbeNode *);
        DeleteCurrent();
        UnlinkCurrent();
        DestroyList();
        InsertAfterCurrent(ProbeNode *);
        long TestBasicInsertion(ProbeNode *);
        Rotate();
        Duplicate(TourClass &);
        InsertLeastCost(ProbeNode *);
        InsertLeastCostFromPool(TourClass &);
        InsertLeastCostByMarker(TourClass &);
        InsertLeastCostWithModifier(ProbeNode *, long);
        QuickInsertLeastCost(ProbeNode *, long);
        ~TourClass();
        Output(ofstream &);
        List(ofstream &);
};

TourClass::TourClass()
{
        DataList = NULL;
        Cost = 0;
}

TourClass::Output(ofstream &OutStream)
{
        ProbeNode *Start = this->DataList;

        this->DataList->DataPointer->Output(OutStream);
        Rotate();
        while (Start != this->DataList)
        {
                this->DataList->DataPointer->Output(OutStream);
                Rotate();
        }
}

TourClass::List(ofstream &OutStream)
{
        ProbeNode *Start = this->DataList;

        this->DataList->DataPointer->List(OutStream);
        Rotate();
        while (Start != this->DataList)
        {
                this->DataList->DataPointer->List(OutStream);
                Rotate();
        }
}

TourClass::Duplicate(TourClass &TestTour)
{
        ProbeNode *Duplicate;
        ProbeNode *Start;

   this->DestroyList();
        Start = TestTour.DataList;
   Duplicate = new ProbeNode;
   Duplicate->Copy(TestTour.DataList);
        InitializeTour(Duplicate);
   TestTour.Rotate();

   while (Start!=TestTour.DataList)
```

```
        {
        Duplicate = new ProbeNode;
      Duplicate->Copy(TestTour.DataList);
      InsertAfterCurrent(Duplicate);
              TestTour.Rotate();
      Rotate();
        }
   return(TRUE);
}

TourClass::InsertAfterCurrent(ProbeNode *NewNode)
{
        ProbeNode *Link;

        if (DataList==NULL)
        {
                InitializeTour(NewNode);
                return(TRUE);
        }
        Link = DataList->Next;
        DataList->Next = NewNode;
        Link->Previous = NewNode;
        NewNode->Next = Link;
        NewNode->Previous = DataList;
        Cost = Cost - DataList->NextCost;
        DataList->NextCost = DataList->Distance(NewNode);
        Cost = Cost + DataList->NextCost;
        NewNode->NextCost = NewNode->Distance(Link);
        Cost = Cost + NewNode->NextCost;
}

long
TourClass::TestBasicInsertion(ProbeNode *NewNode)
{
        long TestCost;

        TestCost = Cost - DataList->NextCost;
        TestCost += DataList->Distance(NewNode);
   TestCost += NewNode->Distance(DataList->Next);
   return(TestCost);
}

TourClass::Rotate()
{
        DataList = DataList->Next;
}

TourClass::InsertLeastCost(ProbeNode *NewNode)
{
        ProbeNode *Start;
        ProbeNode *BestPlace;
   long TestCost, BestCost;

   Start = DataList;
        BestPlace = DataList;
   TestCost = TestBasicInsertion(NewNode);
        BestCost = TestCost;
        Rotate();
   while (DataList!=Start)
        {
                TestCost = TestBasicInsertion(NewNode);
      if (TestCost<BestCost)
      {
        BestCost = TestCost;
          BestPlace = DataList;
      }
      Rotate();
   }
        DataList = BestPlace;
   InsertAfterCurrent(NewNode);
}
```

```
TourClass::InsertLeastCostByMarker(TourClass &Source)
{
      // searches Source for undone ones,
   // adds the closest one to the new tour
      ProbeNode *Start;
   ProbeNode *BestPlace, *BestAdd;
   ProbeNode *NewNode, *Done;
      long TestCost, BestCost;

      Start = DataList;
      BestPlace = DataList;
   NewNode = Source.DataList;
      Done = Source.DataList;
   TestCost = TestBasicInsertion(NewNode);
   BestCost = TestCost;
   Rotate();
      Source.Rotate();
   long totalmarker = 0;
       while(Source.DataList!=Done);
   {
               NewNode = Source.DataList;
          while (DataList!=Start && !NewNode->marker)
               {
                        TestCost = TestBasicInsertion(NewNode);
                        if (TestCost<BestCost)
             {
                                 BestAdd = NewNode;
               BestCost = TestCost;
                 BestPlace = DataList;
                        }
                        Rotate();
                        totalmarker = 1;
          }
      Source.Rotate();
   }
   NewNode = new ProbeNode;
   NewNode->Copy(BestAdd);
   BestAdd->marker = 1;
      DataList = BestPlace;
   InsertAfterCurrent(NewNode);
      return(totalmarker);
}

TourClass::InsertLeastCostFromPool(TourClass &Source)
{
      // searches Source for undone ones,
   // adds the closest one to the new tour
      ProbeNode *Start;
   ProbeNode *BestPlace, *BestAdd;
   ProbeNode *NewNode, *Done;
   long TestCost, BestCost;

      BestAdd = NULL;
      BestPlace = NULL;
      NewNode = Source.DataList;
      Done = Source.DataList;
      TestCost = 1000;
      BestCost = TestCost;
      Source.Rotate();
      long totalmarker = 0;
      while(Source.DataList!=Done)
      {
            NewNode = Source.DataList;
            //cout << NewNode->DataPointer->Probe << TAB << NewNode->marker << endl;
            if (!NewNode->marker)
            {
                  TestCost = TestBasicInsertion(NewNode);
                  if (TestCost<BestCost)
                  {
                        BestAdd = NewNode;
```

```
                        BestCost = TestCost;
                    }
                    totalmarker = 1;
                }
                Source.Rotate();
        }
        if (totalmarker && BestAdd)
        {
                NewNode = new ProbeNode;
                NewNode->Copy(BestAdd);
                BestAdd->marker = 1;
                InsertAfterCurrent(NewNode);
                cout << NewNode->DataPointer->Probe << endl;
                Rotate(); // go to NewNode as the favorite
        }
        return(totalmarker);
}

TourClass::InsertLeastCostWithModifier(ProbeNode *NewNode, long ModMax)
{
        ProbeNode *Start;
        ProbeNode *BestPlace;
        long BestModifier;
        long TestCost, BestCost;

        Start = DataList;
        BestPlace = DataList;
        TestCost = TestBasicInsertion(NewNode);
        BestCost = TestCost;
        BestModifier = 0;

        for (long ModLoop = 0; ModLoop<ModMax; ModLoop++)
        {
                NewNode->DataPointer->SynthModifier = ModLoop;
                Start = DataList;
                Rotate();
                while (DataList!=Start)
                {
                        TestCost = TestBasicInsertion(NewNode);
                        if (TestCost<BestCost)
                        {
                                BestCost = TestCost;
                                BestPlace = DataList;
                                BestModifier = ModLoop;
                        }
                        Rotate();
                }
        }
        DataList = BestPlace;
        NewNode->DataPointer->SynthModifier = BestModifier;
        InsertAfterCurrent(NewNode);
}

TourClass::QuickInsertLeastCost(ProbeNode *NewNode, long SearchLevel)
{
        ProbeNode *Start;
        ProbeNode *BestPlace;
        long TestCost, BestCost;

        Start = DataList;
        BestPlace = DataList;
        TestCost = TestBasicInsertion(NewNode);
        BestCost = TestCost;

        Start = DataList;
        Rotate();
        long counter = 0;
        NewNode->DataPointer->SynthModifier = 0;
        while (DataList!=Start && counter<SearchLevel)
        {
                TestCost = TestBasicInsertion(NewNode);
```

```
                if (TestCost<BestCost)
                {
                        BestCost = TestCost;
                        BestPlace = DataList;
                }
                Rotate();
                counter++;
        }
        DataList = BestPlace;
        InsertAfterCurrent(NewNode);
}

TourClass::DeleteCurrent()
{
        ProbeNode *Link;
        ProbeNode *Del;

        Link = DataList->Next;
    Del = DataList;
        Cost = Cost - DataList->NextCost;
        Cost = Cost - DataList->Previous->NextCost;
    if (Link==DataList)
    {
        DataList = NULL;
      delete Del;
                Cost = 0;
       return(TRUE);
        }
    Link->Previous = DataList->Previous;
    Link->Previous->Next = DataList->Next;
    DataList = Link;
        delete Del;
    Link->Previous->NextCost = Link->Previous->Distance(Link);
    Cost = Cost + Link->Previous->NextCost;
    return(TRUE);
}

TourClass::UnlinkCurrent()
{
        ProbeNode *Link;
    ProbeNode *Del;

        Link = DataList->Next;
    Del = DataList;
    Cost = Cost - DataList->NextCost;
        Cost = Cost - DataList->Previous->NextCost;
    if (Link==DataList)
    {
        DataList = NULL;
                Cost = 0;
       return(TRUE);
    }
    Link->Previous = DataList->Previous;
        Link->Previous->Next = DataList->Next;
    DataList = Link;
        Link->Previous->NextCost = Link->Previous->Distance(Link);
    Cost = Cost + Link->Previous->NextCost;
        return(TRUE);
}

TourClass::InitializeTour(ProbeNode *Test)
{
        DestroyList();
        DataList = Test;
        Test->Initialize();
        Cost = Test->ZeroCost();
        Test->NextCost = 0;
        return(TRUE);
}

TourClass::DestroyList()
```

```
{
        while (DataList!=NULL)
                DeleteCurrent();
}


TourClass::~TourClass()
{
        DestroyList();
}

class TSPClass{
public:
        TourClass DataSet;
        TourClass BestTour;
        TourClass CurrentTour;
        TSPClass();
        ~TSPClass();
        LoadData(string);
        LoadMutantData(string, long);
        LoadMisMatchData(string, long);
        LoadMisMatchFluffData(string, long);
        LoadMisMatchFastFluffData(string, long);
        LoadExpressionData(string, long);
        LoadSingleExpressionData(string);
        LoadExpressionDataByUnit(string, long, long);
        LoadExpressionFluffData(string, long);
        LoadChipData(string);
        GenerateTourByInsertion();
        GenerateTourByInsertionAndDeletion();
        GenerateTourByClosestInsertion();
        GenerateTourByClosestPool();
        GenerateTourByInsertionWithModifier(long);
        GenerateQuickTourByInsertion(long);
        ImproveTourByReplacement(long);
        OutputTour(string);
        AppendTour(string);
        ListTour(string);
        DestroyData();
//   Geni(int);
};

TSPClass::TSPClass()
{
}

TSPClass::DestroyData()
{
        ProbeNode *DataStart;

        DataStart = DataSet.DataList;
        DataSet.DataList->DestroyData();
        DataSet.Rotate();

   while(DataStart!=DataSet.DataList)
   {
          DataSet.DataList->DestroyData();
              DataSet.Rotate();
   }
        return(TRUE);
}

TSPClass::~TSPClass()
{
        //DestroyData();
}

TSPClass::GenerateTourByInsertion()
{
        ProbeNode *DataStart;
        ProbeNode *TempData;
```

```
        DataStart = DataSet.DataList;
        TempData = new ProbeNode;
        TempData->Copy(DataSet.DataList);
        BestTour.InitializeTour(TempData);
        DataSet.Rotate();

        long counter = 0;
        while(DataStart!=DataSet.DataList)
        {
                TempData = new ProbeNode;
                TempData->Copy(DataSet.DataList);
                //cout << DataSet.DataList->DataPointer->Probe << TAB << counter << TAB;
                BestTour.InsertLeastCost(TempData);
                //cout << BestTour.Cost << endl;
                DataSet.Rotate();
                if (counter%100==0)
                        cout << counter << endl;
                counter++;
        }
        return(TRUE);
}

TSPClass::GenerateTourByInsertionAndDeletion()
{
        ProbeNode *DataStart;
        ProbeNode *TempData;


        DataStart = DataSet.DataList;
        if (DataSet.DataList->marker>0)
                DataSet.Rotate();
        while (DataSet.DataList->marker>0 && DataSet.DataList!=DataStart)
        {
                DataSet.Rotate();
        }
        if (DataSet.DataList->marker>0)
                return(FALSE);
        DataStart = DataSet.DataList;
        DataStart->marker = 1;

        TempData = new ProbeNode;
        TempData->Copy(DataSet.DataList);
        BestTour.InitializeTour(TempData);

        long Unit = TempData->DataPointer->Unit;

        DataSet.Rotate();

        long counter = 0;
        while(DataStart!=DataSet.DataList)
        {
                if (DataSet.DataList->DataPointer->Unit==Unit && DataSet.DataList->marker<1)
                {
                        TempData = new ProbeNode;
                        TempData->Copy(DataSet.DataList);
                        cout << DataSet.DataList->DataPointer->Name << TAB << DataSet.DataList-
>DataPointer->Unit << TAB << counter << endl;
                        BestTour.InsertLeastCost(TempData);
                        DataSet.DataList->marker = 1;
                //cout << BestTour.Cost << endl;
                }
                DataSet.Rotate();
                counter++;
                if (counter%1000==0)
                        cout << counter << endl;
        }
        return(TRUE);
}
```

```
TSPClass::GenerateTourByClosestInsertion()
{
        ProbeNode *DataStart;
    ProbeNode *TempData;

        DataStart = DataSet.DataList;
    TempData = new ProbeNode;
        TempData->Copy(DataSet.DataList);
    DataSet.DataList->marker = 1; // set on this course
        BestTour.InitializeTour(TempData);

        long notdone = 1;
        long counter = 0;
    while (notdone)
        {
                notdone = BestTour.InsertLeastCostByMarker(DataSet);
       if (counter%100==0)
              cout << counter << endl;
                counter++;
    }
        return(TRUE);
}

TSPClass::GenerateTourByClosestPool()
{
        ProbeNode *DataStart;
        ProbeNode *TempData;

        DataStart = DataSet.DataList;
        TempData = new ProbeNode;
        TempData->Copy(DataSet.DataList);
        DataSet.DataList->marker = 1; // set on this course
        BestTour.InitializeTour(TempData);

        long notdone = 1;
        long counter = 0;
        while (notdone)
        {
                notdone = BestTour.InsertLeastCostFromPool(DataSet);
                counter++;
        }
        return(TRUE);
}

TSPClass::GenerateTourByInsertionWithModifier(long ModMax)
{
        ProbeNode *DataStart;
        ProbeNode *TempData;

        DataStart = DataSet.DataList;
        TempData = new ProbeNode;
        TempData->Copy(DataSet.DataList);
        BestTour.InitializeTour(TempData);
        DataSet.Rotate();

        long counter = 0;
        while(DataStart!=DataSet.DataList)
        {
                TempData = new ProbeNode;
                TempData->Copy(DataSet.DataList);
                BestTour.InsertLeastCostWithModifier(TempData, ModMax);
                if (counter%100==0)
                {
                        cout << DataSet.DataList->DataPointer->Probe << TAB << counter << TAB;
                        cout << BestTour.Cost << endl;
                }
                DataSet.Rotate();
                counter++;
        }
        return(TRUE);
}
```

```
TSPClass::GenerateQuickTourByInsertion(long SearchLevel)
{
        ProbeNode *DataStart;
        ProbeNode *TempData;

        DataStart = DataSet.DataList;
        TempData = new ProbeNode;
        TempData->Copy(DataSet.DataList);
        BestTour.InitializeTour(TempData);
        DataSet.Rotate();

        long counter = 0;
        while(DataStart!=DataSet.DataList)
        {
                TempData = new ProbeNode;
                TempData->Copy(DataSet.DataList);
                BestTour.QuickInsertLeastCost(TempData, SearchLevel);
                if (counter%100==0)
                {
                        cout << DataSet.DataList->DataPointer->Probe << TAB << counter << TAB;
                        cout << BestTour.Cost << endl;
                }
                DataSet.Rotate();
                counter++;
        }
        return(TRUE);
}

TSPClass::ImproveTourByReplacement(long ReplaceSize)
{
        ProbeNode *DataStart;
        ProbeNode *TempData;


   long counter = 0;
        while(counter<ReplaceSize)
        {
                TempData = BestTour.DataList;
                if (TempData->marker<1)
      {
                        TempData->marker = 1;
                        BestTour.UnlinkCurrent();
             cout << TempData->DataPointer->Probe << TAB << counter << TAB;
             BestTour.InsertLeastCost(TempData);
          cout << BestTour.Cost << endl;
                }
      else
                        BestTour.Rotate();
      counter++;
        }
        return(TRUE);
}

TSPClass::LoadData(string FileName)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
   string TestString;
   ProbeNode *NewNode;
        float test; // soak up number
        string junkstring;
```

```
        while (!ExpStream.eof())
        {
                ExpStream >> TestString >> junkstring >> test;
                if (TestString.length()>1)
      {
                        NewNode = new ProbeNode;
                        NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Original = TestString;
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->Synthesize();
                        this->DataSet.InsertAfterCurrent(NewNode);
                        //cout << TestString << endl;
                }
        }
        ExpStream.close();
        cout << DataSet.Cost << endl;
}

TSPClass::LoadMutantData(string FileName, long MutateValue)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
        string TestString;
        ProbeNode *NewNode;
        float test; // soak up number

        while (!ExpStream.eof())
        {
                ExpStream >> TestString >> test;
                if (TestString.length()>1)
      {
             NewNode = new ProbeNode;
          NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->SynthesizeMutant(MutateValue);
        this->DataSet.InsertAfterCurrent(NewNode);
                }
        }
        ExpStream.close();
   cout << DataSet.Cost << endl;
}

TSPClass::LoadMisMatchData(string FileName, long MutateValue)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
   string TestString;
        ProbeNode *NewNode;
        float test; // soak up number

        while (!ExpStream.eof())
        {
```

```
                ExpStream >> TestString >> test;
                if (TestString.length()>1)
                {
                        NewNode = new ProbeNode;
            NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->SynthesizeMisMatch(MutateValue);
                        this->DataSet.InsertAfterCurrent(NewNode);
        }
        }
        ExpStream.close();
    cout << DataSet.Cost << endl;
}

TSPClass::LoadMisMatchFluffData(string FileName, long MutateValue)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
    string TestString;
        ProbeNode *NewNode;
        float test; // soak up number

        while (!ExpStream.eof())
        {
                ExpStream >> TestString >> test;
      if (TestString.length()>1)
                {
              NewNode = new ProbeNode;
                        NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Probe = TestString;
           NewNode->DataPointer->SynthesizeFluff(MutateValue);
        this->DataSet.InsertAfterCurrent(NewNode);
        }
        }
        ExpStream.close();
    cout << DataSet.Cost << endl;
}

TSPClass::LoadMisMatchFastFluffData(string FileName, long MutateValue)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
    string TestString;
        ProbeNode *NewNode;
        float test; // soak up number

        while (!ExpStream.eof())
        {
                ExpStream >> TestString >> test;
      if (TestString.length()>1)
                {
              NewNode = new ProbeNode;
                        NewNode->DataPointer = new ProbeSynth;
```

```
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->SynthesizeFastFluff(MutateValue);
                        this->DataSet.InsertAfterCurrent(NewNode);
                }
                }
                ExpStream.close();
                cout << DataSet.Cost << endl;
}

TSPClass::LoadExpressionData(string FileName, long MutateValue)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
        string TestString;
        ProbeNode *NewNode;
        float test; // soak up number
        long facevalue;
        long Unit, Atom;
        string Name, Rename;
        string Junk;
        long counter = 0;
        ExpStream >> Junk >> Junk >> Junk >> Junk >> Junk >> Junk;
        while (!ExpStream.eof())
        {
                ExpStream >> facevalue >> TestString >> Name >> Rename >> Unit >> Atom;
                if (TestString.length()>1)
                {
                        NewNode = new ProbeNode;
                        NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Original = TestString;
                        transform(TestString);
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->Name = Name;
                        NewNode->DataPointer->Rename = Rename;
                        NewNode->DataPointer->Location = facevalue;
                        NewNode->DataPointer->Unit = Unit;
                        NewNode->DataPointer->Atom = Atom;
                        NewNode->DataPointer->SynthesizeMisMatch(MutateValue);
                        this->DataSet.InsertAfterCurrent(NewNode);
                }
                counter++;
                if (counter%100==0)
                        cout << counter << TAB << Name << TAB << TestString << endl;
        }
        ExpStream.close();
        cout << DataSet.Cost << endl;
}
TSPClass::LoadSingleExpressionData(string FileName)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        int probeloop = 0;
        string TestString;
        ProbeNode *NewNode;
```

```
        float test; // soak up number
        int facevalue;
        int Unit, Atom;
        string Name;
        string Junk;
        int counter = 0;
        ExpStream >> Junk >> Junk >> Junk >> Junk >> Junk >> Junk;
        while (!ExpStream.eof())
        {
                ExpStream >> facevalue >> TestString >> Name >> Junk >> Unit >> Atom;
                if (TestString.length()>1)
                {
                        NewNode = new ProbeNode;
                        NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Original = TestString;
                        transform(TestString);
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->Name = Name;
                        NewNode->DataPointer->Location = facevalue;
                        NewNode->DataPointer->Unit = Unit;
                        NewNode->DataPointer->Atom = Atom;
                        NewNode->DataPointer->Synthesize();
                        this->DataSet.InsertAfterCurrent(NewNode);
                }
                counter++;
                if (counter%100==0)
                        cout << counter << TAB << Name << TAB << TestString << endl;
        }
        ExpStream.close();
        cout << DataSet.Cost << endl;
}

TSPClass::LoadExpressionDataByUnit(string FileName, long MutateValue, long UnitLimit)
{
        // build the basic datalist
        ifstream ExpStream;
        ExpStream.open(FileName.c_str(), ios::in);

        if (ExpStream.bad())
        {
                cout << endl << "Experimental data file specified does not exist!";
                cout << endl << endl;
                exit(0);
        }
        long probeloop = 0;
        string TestString;
        ProbeNode *NewNode;
        float test; // soak up number
        long facevalue;
        long Unit, Atom;
        string Name;
        string Junk;
        long counter = 0;
        while (!ExpStream.eof())
        {
                ExpStream >> facevalue >> TestString >> Name >> Junk >> Unit >> Atom;
                if (TestString.length()>1 && Unit==UnitLimit)
                {
                        NewNode = new ProbeNode;
                        NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Original = TestString;
                        transform(TestString);
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->Name = Name;
                        NewNode->DataPointer->Location = facevalue;
                        NewNode->DataPointer->Unit = Unit;
                        NewNode->DataPointer->Atom = Atom;
                        NewNode->DataPointer->SynthesizeMisMatch(MutateValue);
                        this->DataSet.InsertAfterCurrent(NewNode);
                }
                counter++;
```

```
      if (counter%100==0)
       cout << counter << TAB << Name << TAB << TestString << endl;
       }
       ExpStream.close();
       cout << DataSet.Cost << endl;
}

TSPClass::LoadExpressionFluffData(string FileName, long MutateValue)
{
       // build the basic datalist
       ifstream ExpStream;
       ExpStream.open(FileName.c_str(), ios::in);

       if (ExpStream.bad())
       {
              cout << endl << "Experimental data file specified does not exist!";
              cout << endl << endl;
              exit(0);
       }
       long probeloop = 0;
       string TestString;
       ProbeNode *NewNode;
       float test; // soak up number
       long facevalue;
       long Unit, Atom;
       string Junk;
       string Name;
       long counter = 0;
       while (!ExpStream.eof())
       {
       ExpStream >> facevalue >> TestString >> Name;
      if (TestString.length()>1)
              {
                     NewNode = new ProbeNode;
         NewNode->DataPointer = new ProbeSynth;
                     transform(TestString);
            NewNode->DataPointer->Probe = TestString;
        NewNode->DataPointer->Name = Name;
                     NewNode->DataPointer->Location = facevalue;
                     NewNode->DataPointer->SynthesizeFastFluff(MutateValue);
                     this->DataSet.InsertAfterCurrent(NewNode);
              }
      counter++;
      if (counter%100==0)
       cout << counter << TAB << Name << TAB << TestString << endl;
       }
       ExpStream.close();
       cout << DataSet.Cost << endl;
}

TSPClass::LoadChipData(string FileName)
{
       // build the basic datalist
       ifstream ExpStream;
       ExpStream.open(FileName.c_str(), ios::in);

       if (ExpStream.bad())
       {
              cout << endl << "Experimental data file specified does not exist!";
              cout << endl << endl;
              exit(0);
       }
       long probeloop = 0;
       string TestString;
   ProbeNode *NewNode;
       long test; // soak up number

       while (!ExpStream.eof())
       {
              ExpStream >> TestString >> test;
      if (TestString.length()>1)
```

```
        {
                NewNode = new ProbeNode;
            NewNode->DataPointer = new ProbeSynth;
                        NewNode->DataPointer->Probe = TestString;
                        NewNode->DataPointer->SynthesizeMutant(test-1);
                        this->DataSet.InsertAfterCurrent(NewNode);
                }
        }
        ExpStream.close();
        cout << DataSet.Cost << endl;
}

TSPClass::OutputTour(string FileName)
{
        ofstream OutStream;
        OutStream.open(FileName.c_str(), ios::out);

        this->BestTour.Output(OutStream);

        OutStream.close();
}

TSPClass::ListTour(string FileName)
{
        ofstream OutStream;
        OutStream.open(FileName.c_str(), ios::out);

        this->BestTour.List(OutStream);

        OutStream.close();
}

TSPClass::AppendTour(string FileName)
{
        ofstream OutStream;
        OutStream.open(FileName.c_str(), ios::app);

        this->BestTour.Output(OutStream);

        OutStream.close();
}

static void do_gematch(long Value)
{
        TSPClass Example;

        Example.LoadMisMatchFluffData("gematch.prb", 10-1);
        Example.GenerateTourByInsertionWithModifier(Value);
        Example.OutputTour("gematch.flf");
}

static void do_yematch()
{
        TSPClass Example;

        Example.LoadExpressionData("yematch.prb", 10-1);
        Example.GenerateTourByInsertionWithModifier(1);
        Example.OutputTour("yematch.tsp");
}

static void do_yematch_local(long UnitMatch)
{
        TSPClass Example;

        Example.LoadExpressionDataByUnit("yematch.prb", 10-1, UnitMatch);
        Example.GenerateTourByInsertionWithModifier(1);
        Example.AppendTour("yematch.lsp");
}

static void do_yematch_local_pool(long UnitMatch)
{
```

```
        TSPClass Example;

        Example.LoadExpressionDataByUnit("ematch.prb", 10-1, UnitMatch);
        Example.GenerateTourByClosestPool();
        Example.AppendTour("ematch.csp");
}

static void do_hummatch_local(long UnitMatch)
{
        TSPClass Example;

        Example.LoadExpressionData("ha.prb", 13-1);
        Example.GenerateQuickTourByInsertion(2048);
        Example.OutputTour("ha.tsp");

        //while (Example.GenerateTourByInsertionAndDeletion())
                //Example.AppendTour("ha.lsp");
}

static void do_fast_gematch(long Value)
{
        TSPClass Example;

        Example.LoadMisMatchFastFluffData("gematch.prb", 10-1);
        Example.GenerateTourByInsertionWithModifier(Value);
        Example.OutputTour("gefast.flf");
}

static void do_fast_pool_gematch(long Value)
{
        TSPClass Example;

        Example.LoadMisMatchFastFluffData("gematch.tny", 10-1);
        Example.GenerateTourByClosestPool();
        Example.OutputTour("gefast.pl");
}

static void do_mito(long Value)
{
        TSPClass Example;

        Example.LoadMutantData("mt9566.prb", 10-1);
        Example.GenerateTourByInsertionWithModifier(Value);
        //Example.ImproveTourByReplacement(1000);
        string Test;
        Test = "mt9566.";
   char ctest = 'a'+Value;
        Test += ctest;
   Example.OutputTour(Test);
}


static void do_hiv(long Value)
{
        TSPClass Example;

        Example.LoadChipData("hv430a.prb");
        Example.GenerateTourByInsertionWithModifier(Value);
        //Example.ImproveTourByReplacement(1000);
        string Test;
  Test = "hv430.";
        char ctest = 'a'+Value;
   Test += ctest;
        Example.OutputTour(Test);
}

static void do_new_ge()
{
        TSPClass Example;
```

```
        Example.LoadExpressionFluffData("i:\\data\\metrix\\ehubbe\\design\\nmisc\\gem01\\gem01.prb",
10-1);
        Example.GenerateTourByInsertionWithModifier(1);
        Example.OutputTour("gem01.flf");
}


static void do_noise()
{
        TSPClass Example;

        Example.LoadData("noisea8.20");
        Example.GenerateTourByInsertionWithModifier(1);
        Example.ListTour("na8_20.tsp");
}

static void do_noise_two()
{
        TSPClass Example;

        Example.LoadData("c:\\cover\\noisea8.16");
        Example.GenerateTourByInsertionWithModifier(1);
        Example.ListTour("na8_16.tsp");
}

static void do_noise_three()
{
        TSPClass Example;

        Example.LoadData("c:\\cover\\noisea8.18");
        Example.GenerateTourByInsertionWithModifier(1);
        Example.ListTour("na8_18.tsp");
}

static void do_cost_one()
{
        TSPClass Example;

        Example.LoadData("c:\\genius\\testa8.20");
        Example.GenerateTourByInsertionWithModifier(1);
        Example.ListTour("ca8_20.tsp");
}

static void do_cost_two()
{
        TSPClass Example;

        Example.LoadData("c:\\genius\\ca8_20.prb");
        Example.GenerateTourByInsertionWithModifier(1);
        Example.ListTour("cba8_20.tsp");
}

static void do_cost_quick()
{
        TSPClass Example;

        Example.LoadData("c:\\genius\\noise\\ca8_20.rnd");
        Example.GenerateQuickTourByInsertion(1024);
        Example.ListTour("cqa8_20.45");
}

static void do_rat_local(long UnitMatch)
{
        TSPClass Example;

        Example.LoadExpressionData("r:\\alldes\\cdesign\\ter09\\included_probes\\normal_probes.txt",
13 -1 );
        Example.LoadExpressionData("r:\\alldes\\cdesign\\ter09\\included_probes\\sense\\reverse_comp
_seqs.txt", 13 -1);
//      Example.LoadSingleExpressionData("r:\\alldes\\cdesign\\eol0191\\eoshu02.dat");
//      Example.GenerateQuickTourByInsertion(140000);
```

```
        Example.GenerateQuickTourByInsertion(20480);
        Example.OutputTour("r:\\alldes\\cdesign\\ter09\\full_fix.tsp");

        //while (Example.GenerateTourByInsertionAndDeletion())
                //Example.AppendTour("ha.lsp");
}

main()
{
        do_rat_local(0);
//      do_fast_pool_gematch(1);
//      do_yematch();
//      for (long i=1; i<104; i++)
//              do_yematch_local_pool(i);
//      do_gematch(1);
//    do_new_ge();
//      do_noise();
//      do_noise_two();
//      do_noise_three();
//      do_cost_quick();
}
```